The Report Committee for Iat-Kei Ho
Certifies that this is the approved version of the following report:

Fusion-based Hadoop MapReduce Job for Fault Tolerance in Distributed Systems

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

Vijay K. Garg

Zhigang Yi

Fusion-based Hadoop MapReduce Job for Fault Tolerance in Distributed Systems

by

Iat-Kei Ho, B.S.C.S.; B.S.Math.

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2013

Abstract

Fusion-based Hadoop MapReduce Job for Fault Tolerance in Distributed Systems

Iat-Kei Ho, M.S.E.

The University of Texas at Austin, 2013

Supervisor: Vijay K. Garg

Standard recovery solution on a failed task in Hadoop systems is to execute the task again. After retrying for a configured number of times, it is marked as failure. With significant amount of data, complicated Map and Reduce functions, recovering corrupted or unfinished data from a failed job can be more efficient than re-executing the same job. This paper is an extension of [1] by applying fusion-based technique [7][8] in Hadoop MapReduce tasks execution to enhance its fault tolerance. Multiple data sets are executed through Hadoop MapReduce with and without fusion in various pre-defined failure scenarios for comparison. As the complexity of the Map and Reduce function relative to the Recover function increases, it becomes more efficient to utilize fusion and users can tolerate faults by incurring less than ten percent of extra execution time.

Table of Contents

List of Tables

List of Figures

Chapter 1. Introduction

The Apache Hadoop is a software library based on Google's MapReduce [2] and Google File System [3]. It allows data processing tasks and redundant data storage to be distributed across a cluster of computers with automatic retry mechanism when failures are detected. Due to its effectiveness of computing on large data set, other projects are created on top of Hadoop including:

- Cassandra™: scalable multi-master database with no single points of failure.[4]
- HBase™: scalable, distributed database that supports structured data storage for large tables. [4]
- Hive™: data warehouse infrastructure that provides data summarization and ad hoc querying. [4]
- Mahout™: machine learning and data mining library. [4]
- Pig™: high-level data-flow language and execution framework for parallel computation. [4]
- ZooKeeper™: high-performance coordination service for distributed applications. [4]

The focus of this paper is the MapReduce aspect of Hadoop release 0.20.

MAPREDUCE PROGRAMMING MODEL

MapReduce programming model consists of multiple phases in sequence as illustrated in diagram 1. The following is a brief description of each phase:

1. Map: It reads in input as key-value pairs usually with the line number as the key, a line from input data as the value, parses data and writes key-value pairs as output.

2. Sort and Shuffle: It groups the output from Map phase in a sequence ready for Reducer to consume, so all value parts of a pair with the same key are grouped together. Hadoop provides a default implementation and also provides a mechanism for users to customize the comparison and sorting logic.
3. Combine and Partition: Before Map outputs from multiple nodes are sent to other nodes for Reduce phase, Combine reduces values in a node locally to reduce data traveling from one node to another. As some keys may contain more values than others, Partition allows users to fine tune the spread of keys to Reduce nodes in order to balance the load. A default implementation for Partition is also included in Hadoop which utilizes the hashcode of Java objects. A Combiner acts very similar to Reducer and is optional in the process.
4. Reduce: Reduce eventually reads a list of values for a given key and converts them to the desired output.

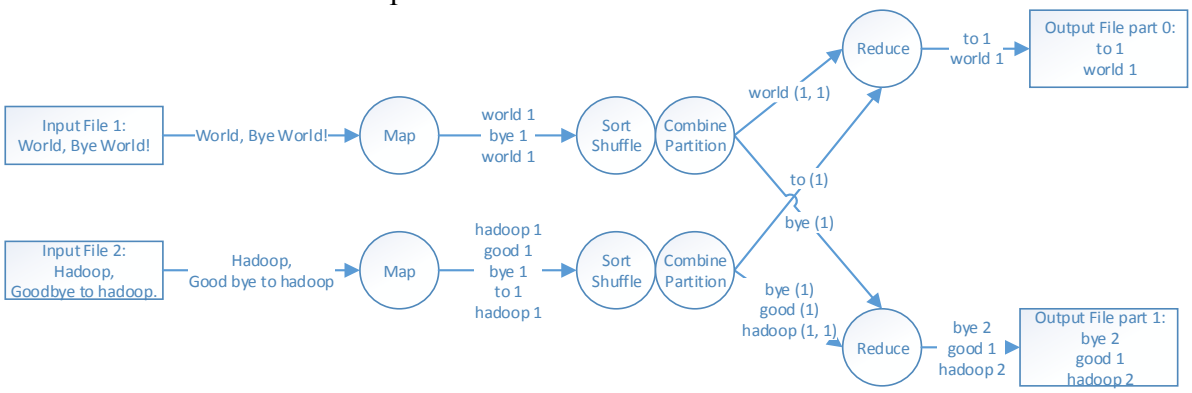

Figure 1. MapReduce flow diagram of a simple word count example.

EXECUTION TRACKING AND FAULTS HANDLING

On top of the data manipulation phases described above, Hadoop MapReduce provides execution tracking which supports a retry mechanism after a process fault and a

process timeout. Each Map or Reduce task is executed within a TaskTracker process and monitored by a JobTracker process. When a TaskTracker fault or timeout is detected by the JobTracker, the same task is created, and another free TaskTracker executes the same task as a fallback. If the job is still in the mapping phase, the free TaskTrackers re-execute all map tasks previously run by the failed TaskTracker [5]. If the job is in the reducing phase, the other TaskTrackers re-execute all reduce tasks that were in progress on the failed TaskTracker [5]. The default number of times for retrying is four, and can be configured differently for Map and Reduce. A node failed after a configured number of times is blacklisted for the job it was executing, and will not be assigned tasks for this job [6].

Chapter 2. Fusion Execution

Extending [1], we combine two keys, and add the combined keys to the overall execution in a MapReduce job trying to increase fault tolerance without repeating part of the job.

EXECUTION FLOW

Since there are multiple ways to implement Fusion-based Hadoop MapReduce job, the following assumptions are made to guide the implementation direction:

1. No shared memory/data between nodes when MapReduce job is running to avoid the overhead of keeping potentially large number of nodes in sync.
2. Fusion will not be implemented in Map phase, hence fault tolerance will not be improved in Map phase, as the system won't get all input data or events if Map phase failed to construct fusion key pairs.
3. Complete set of input data is too large to fit in memory of one node.
4. Complete set of keys is too large to fit in memory of one node.
5. Every key contains a valid result and a missing result implies a fault during Reduce of the given key.
6. Implementation of a fusion Reduce is relatively simple and similar computation-wise comparing to Reduce of a regular key.
7. Creating a new key from two existing keys unique to any other key is relatively trivial.
8. Given a Reduce result of fused key and a result of one parent key, it is simple to compute the result of the other parent key.
9. Fusion execution is started during Map process instead of Reduce process to prevent a Reduce node crash causing fusion not to be started at all.

The whole execution includes four Hadoop jobs working in sequence. Job 1 (fusion key creation) matches two keys together and creates a sequence file with pairs of key-value tuple of the original's job Mapper output. Job 2 (fusion execution) reads the output from Job 1 instead of reading the original input again, computes result for each key including fused key. At the end of the job 2 Mapper, a custom Partitioner is introduced to separate keys in three buckets: fused key, first parent of the fused key, and second parent of the fused key to avoid a Reduce node handling both the parents of a fused key or parent and a fused key making it impossible for recovery. Retry is disabled for job 2 Reduce to leverage data recovery with result created with Fusion. Job 3 (missed key search) computes keys required for recovering error by comparing the fused keys created in Job 1 and results created in Job 2. Job 4 (recover missed key) computes result for error key by executing Recover function on the result from the fused keys and result from one of its parent. The overall MapReduce job setup is described in table B, while the annotation and symbols used are in table A.

J	Hadoop job to be enhanced with fusion
F_M	Map function of J writes (key, value) pair to context
F_R	Reduce function of J writes (key, value) pair to output
D	Input data of J
K	Set of keys created by M through D
K_F	Set of keys created by fusing two keys in K
R_D	Result of F_R with D
R_{FK}	Result of F_R with K_F
F_R	Function to recover R_{FK} to result of a missing key
K_{FV}	Structure with parent of each element in K_F and their corresponding values
K_M	Set of keys missing in R_D
M_{KM}	Hadoop Map file with K_M
S_{KF}	Hadoop Sequence file with K_{FV}
k	Key entered in Map or Reduce function
v	Value entered in Map function or an element in the Reduce function input Iterable
k_M	Key missing in R_D
k_R	Key with value required for recovering k_M

Table A: Symbols and annotation

Job	Map input	Map function	Reduce function
Fusion key creation	D	Invokes F_M	Combine two keys and write key pair and their associated value list to S_{KF}
Fusion execution	S_{KF}	1. Write(k, value list associated to k) 2. Write(other key, value list associated to other key) 3. Construct fused key 4. Write(fused key, value from value list) 5. Write(fused key, value from other value list)	If key in K_F, write (key, F_R(values)) to R_{FK} else, write (key, F_R(values)) to R_D
Missed key search	R_D	Writes (key, empty)	If empty doesn't exist, write(value, key) to MKM
	S_{KF}	Writes (key, value)	
Recover missed key	R_D	k_M = lookup k_R in M_{KM} by key if exists, writes (k_M from M_{KM}, value from R_D for k_R)	Write (key, F_R(resultD, resultFK))
	R_{FK}	k_M = lookup k_R in M_{KM} by key if exists, writes (k_M from M_{KM}, value from R_{FK} for k_R)	

Table B: Fusion-based Hadoop jobs summary

ADVANTAGES OF FUSION BASED HADOOP MAP REDUCE

Fusion based Hadoop Map Reduce improves application fault by recovering missing data from fused result when there is an error for a specific key either due to bad data or Reduce code problem, which cannot be recovered even by executing the job again. It also improves process fault tolerance by skipping the re-run of a potentially expensive Reduce function.

DISADVANTAGES OF FUSION BASED HADOOP MAP REDUCE

Fusion based Hadoop Map Reduce depends on a Recover function which may not be trivial to implement. The extra overhead in execution time during key pair creation and Map/Reduce execution of the newly created set of fused keys can be costly and is added to the overall cost even when there is no error. Since fusion is created by two other keys, if two out of these three keys computation fail, recovery will not be possible. Extra disk space is required to hold key pair information for the second job to execute.

Chapter 3. Evaluation

In [1], a brief description of MapReduce example was introduced. In this paper, the following factors are used to compare a MapReduce job and a Fusion-based MapReduce job: application fault, process fault, data size, complexity of Map and Reduce functions. The basic word count job, a common example, is being evaluated in this exercise. When executing a simple word count job with fusion, the more complicated input data structure, the extra data keys due to fusion, and the extra requirement for partitioning Map output to Reducer have caused the execution to take more than three times the amount of time of execution without fusion even when data set size increased from 147MB to 394MB. This result is relatively consistent across the process error and data error scenarios. Process error is tested by killing a Reduce process. Data error is simulated by throwing exception for a specific data entry. One difference in the output for data error simulation is that the fused result includes the output for the error, but the job without fusion doesn't. The result of these several consistent behavior is displayed in the following tables and figures.

Data set	Name	Data size (MB)	Unique word count
1	p000000010p000010000	147	581656
2	p000010002p000024999	243	838351
3	p000025001p000055000	368	1120947
4	p000055002p000104998	394	1218265

Table C: Test data set from Wikipedia archive with punctuations removed during the Map phase.

Scenario	Added Map complexity	Added Reduce complexity	Key fault	Process fault
1	No	No	No	No
2	No	No	No	Yes
3	No	No	Yes	No

Table D: Evaluation scenarios information

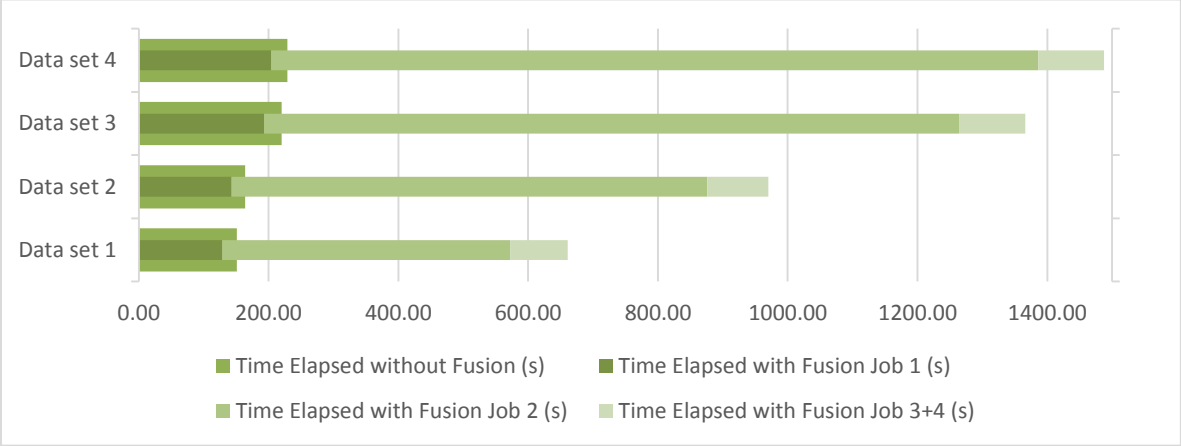

Figure 2: Scenario 1 result

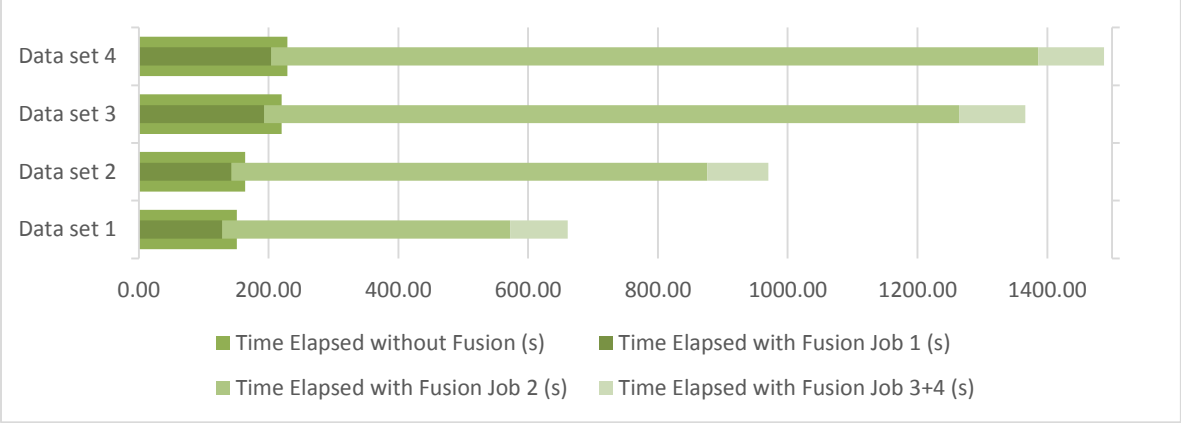

Figure 3: Scenario 2 result

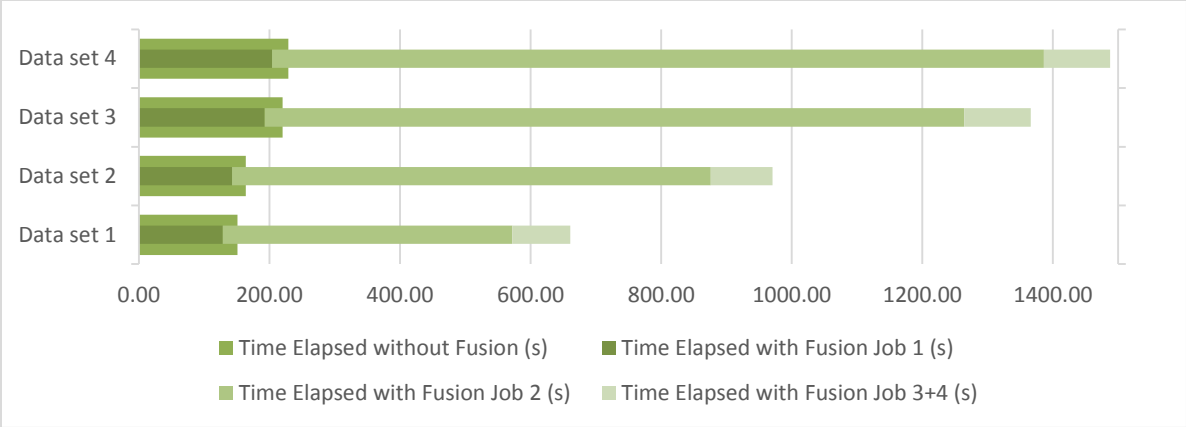

Figure 4: Scenario 3 result

Due to the simplicity of the word count Reducer (summation of a list of 1s) and the word count Mapper (splitting text by space), an extra routine, finding prime numbers between 2000 and 2999 in a loop, is added to simulate a more complicated routine. After increasing the complexity for the Map function and Reduce function, the time overhead for executing with Fusion has significantly improved. The chart below shows the percentage of the extra time added due to fusion relative to the amount of time added in Map and Reduce function.

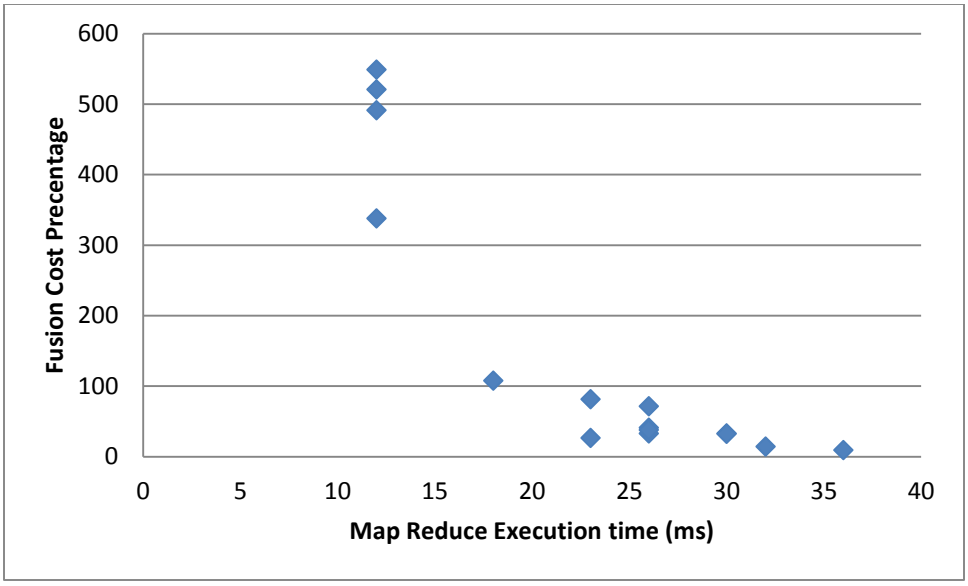

Figure 5: Fusion cost over Map/Reduce function complexity

Chapter 4. Conclusion

Due to the extra development effort, significant overhead in creating matching pair of keys and the extra execution of fused keys, it would not be efficient time-wise if the Map and Reduce routines are relatively simple. As the complexity of the Map and Reduce function relative to the Recover function increases, overhead caused by fusion decreases relative to the overall execution, and users can enjoy its fault tolerance by sacrificing less than ten percent of extra execution time. To avoid the more complicated data structure created in fusion key creation job, storing the fusion key mapping in memory and Hadoop map file were also evaluated. These efforts were aborted either by exceeding heap space or execution being too slow caused by parsing Hadoop Map file for each Map function execution. Other future next steps include:

1. Remove the assumption of shared memory and evaluate Zookeeper™ as a medium to store shared keys.
2. Fuse more than two keys to reduce the total number of keys from 1.5 times of total keys to 1.33 times or 1.25 times of total keys with three keys and four keys during fusion respectively.
3. Leverage the new MapReduce API in YARN to potentially reduce the overhead of the extra jobs.

Appendix

The following is the source code involved in executing word count with and without fusion. WordCount.java contains the driver, Map, and Reduce class for word count without fusion. WordCountFused.java is the driver for word count with fusion. It invokes FusionKeyCreation3 for Job 1, FusionExection3 for Job 2 which depends on FusionPartitioner, MissedKeySearch3 for Job 3, and DefuseMissedKeysWithoutMapFile for Job 4.

WORDCOUNT.JAVA

```
package fusion.hadoop;

import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import fusion.hadoop.fusionexecution.FusionExecution3;


/**
 * Main driver class with embedded Mapper and Reducer class for doing
word count without fusion
 */
public class WordCount
{
    /**
     * Mapper for writing 1 for each word after tokenizer
     */
    public static class WordCountMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
        private Text word = new Text();
        private final static IntWritable one = new IntWritable(1);

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
```

```
            /// invokes a function which simulates a more complicated
routine
            FusionExecution3.FusionExecutionReducer.mapCompute(value,
null);
            for (String token :
WordCountFused.WordCountMapper.map(value)) {
                word.set(token);
                context.write(word, inputMapper(value));
            }
        }

        public static IntWritable inputMapper(Text key)
        {
            return one;
        }
    }

    /**
     * Reducer for writing the sum of the input 1s which results in
counting
     */
    public static class WordCountReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context)
                        throws IOException, InterruptedException {
            /// simulation for application fault
            /// if (key.toString().compareTo("a") == 0) throw new
NullPointerException("Fail to reduce.");

            int sum = 0;
            for (IntWritable value : values) {
                sum += value.get();
            }

            /// invokes a function which simulates a more complicated
routine
            FusionExecution3.FusionExecutionReducer.reduceCompute(key,
values);
            context.write(key, new IntWritable(sum));
        }
    }

    /**
     * Main driver for word count job without fusion
     * @param args [0]: directory of the input files
     * args [1]: directory of the output files
     * @throws IOException
     * @throws InterruptedException
     * @throws ClassNotFoundException
     */
```

```
    public static void main( String[] args ) throws IOException,
InterruptedException, ClassNotFoundException
    {
        System.out.println("\n*** WordCount start...");
        long msStart = System.currentTimeMillis();
        if (args.length != 2)
        {
            System.err.println("Usage: WordCount <input path> <output
path>");
            System.exit(-1);
        }

        Configuration conf = new Configuration();
        conf.setInt("mapreduce.job.reduces",
FusionConfiguration.NUM_OF_REDUCERS);

        FileSystem fs = FileSystem.get(conf);
        fs.delete(new Path(args[1]), true);

        Job job = Job.getInstance(conf);
        job.setJarByClass(WordCount.class);
        job.setJobName("Word Count");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(WordCountMapper.class);
        job.setReducerClass(WordCountReducer.class);
        job.setNumReduceTasks(FusionConfiguration.NUM_OF_REDUCERS);

        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        boolean result = job.waitForCompletion(true);
        long msEnd = System.currentTimeMillis();
        System.out.println("\n*** Total elapsed: " + (msEnd - msStart)
+ "ms");
        System.exit(result ? 0 : 1);
    }
}
```

WORDCOUNTFUSED.JAVA

```
package fusion.hadoop;

import java.io.IOException;
import java.net.URISyntaxException;
import java.util.ArrayList;
import java.util.StringTokenizer;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
```

```
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import fusion.hadoop.defuseMissedKeys.DefuseMissedKeysWithoutMapFile;
import fusion.hadoop.fusionexecution.FusionExecution3;
import fusion.hadoop.fusionkeycreation.FusionKeyCreation3;
import fusion.hadoop.fusionkeycreation.MapFileParser;
import fusion.hadoop.missedKeySearch.MissedKeySearch3;

/**
 * Main driver class with embedded Mapper and Reducer class for doing
word count with fusion
 * It kicks off all four jobs in sequence
 */
public class WordCountFused
{
    /**
     * Mapper for writing 1 for each word after tokenizer
     */
    public static class WordCountMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
        private Text word = new Text();
        private final static IntWritable one = new IntWritable(1);

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            StringTokenizer tokenizer = new
StringTokenizer(value.toString());
            while (tokenizer.hasMoreTokens()) {
                word.set(tokenizer.nextToken());
                context.write(word, one);
            }
        }

        public static String charsToSkip = " \n\r\t\f~`!@#$%^&*()_-
+={}[]|\\\":;'<>,.?/";
        public static String[] patternsToSkip = {
            "&amp;",
            "&lt;",
            "&quot;",
            "&gt;",
            "nbsp;",
            "dash;",
        };


        public static ArrayList<String> map(Text value)
                throws IOException, InterruptedException {
            String strValue = value.toString();

            for (String pattern : patternsToSkip) {
```

```
                strValue = strValue.replaceAll(pattern, "
").toLowerCase();
            }

            ArrayList<String> keys = new ArrayList<String>();
            StringTokenizer tokenizer = new StringTokenizer(strValue,
charsToSkip, false);
            while (tokenizer.hasMoreTokens()) {
                keys.add(tokenizer.nextToken());
            }
            return keys;
        }
    }

    /**
     * Reducer for writing the sum of the input 1s which results in
counting
     */
    public static class WordCountReducer
    extends Reducer<Text, IntWritable, Text, IntWritable> {

        @Override
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context)
                        throws IOException, InterruptedException {

            int sum = 0;
            for (IntWritable value : values) {
                sum += value.get();
            }
            context.write(key, new IntWritable(sum));
        }
    }

    public static void main( String[] args ) throws IOException,
InterruptedException, ClassNotFoundException, URISyntaxException
    {
        System.out.println("\n*** WordCountFusion start...");
        System.out.println("\n*** computation test...");
        long msStart1 = System.currentTimeMillis();
        FusionExecution3.FusionExecutionReducer.mapCompute(null, null);
        System.out.println("\n*** map compute elapsed: " +
(System.currentTimeMillis() - msStart1) + "ms");
        long msStart2 = System.currentTimeMillis();
        FusionExecution3.FusionExecutionReducer.reduceCompute(null,
null);
        System.out.println("\n*** reduce compute elapsed: " +
(System.currentTimeMillis() - msStart2) + "ms");

        if (args.length != 2)
        {
            System.err.println("Usage:: WordCount <input path> <output
path>");
```

```
            System.exit(-1);
        }
        long msStart = System.currentTimeMillis();
        final String fusionKeyMapPath =
"/user/peter/fusion/FusionKeyMap";
        final String missingKeySearchResult =
"/user/peter/fusion/MissingKeySearchResult";
        final String missingKeyDefuseResult =
"/user/peter/fusion/MissingKeyDefuseResult";
        final String executionResultPath = args[1];
        final String inputPath = args[0];

        long msTemp = System.currentTimeMillis();
        int status = 0;
        status = FusionKeyCreation3.main(inputPath, fusionKeyMapPath);
        System.out.println("*** Job elapsed: " +
(System.currentTimeMillis() - msTemp) + "ms\n"); msTemp =
System.currentTimeMillis();
        if (status == 0) status =
FusionExecution3.main(fusionKeyMapPath, executionResultPath);
        System.out.println("*** Job elapsed: " +
(System.currentTimeMillis() - msTemp) + "ms\n"); msTemp =
System.currentTimeMillis();
        if (status == 0) status =
MissedKeySearch3.main(executionResultPath + "/result-r-*",
MapFileParser.PATH + "/fusionkeyvalue-r-*", missingKeySearchResult);
        System.out.println("*** Job elapsed: " +
(System.currentTimeMillis() - msTemp) + "ms\n"); msTemp =
System.currentTimeMillis();
        if (status == 0) status =
DefuseMissedKeysWithoutMapFile.main(executionResultPath + "/result-r-
*", executionResultPath, missingKeySearchResult,
missingKeyDefuseResult);
        System.out.println("*** Job elapsed: " +
(System.currentTimeMillis() - msTemp) + "ms\n"); msTemp =
System.currentTimeMillis();
        long msEnd = System.currentTimeMillis();
        System.out.println("\n*** Total elapsed: " + (msEnd - msStart)
+ "ms");
        System.exit(status);
    }

}
```

TEXTPAIR.JAVA

```
package fusion.hadoop;

import java.io.*;
```

```java
import org.apache.hadoop.io.*;

/**
 * A Writable for storing a pair of keys
 */
public class TextPair implements WritableComparable<TextPair> {

    private Text first;
    private Text second;

    public TextPair() {
        first = new Text();
        second = new Text();
    }

    public TextPair(Text first, Text second) {
        set(first, second);
    }

    public void set(Text first, Text second) {
        this.first = first;
        this.second = second;
    }

    public void set(String first, String second) {
        this.first.set(first);
        this.second.set(second);
    }

    public Text getFirst() {
        return first;
    }

    public Text getSecond() {
        return second;
    }

    public void write(DataOutput out) throws IOException {
        first.write(out);
        second.write(out);
    }

    public void readFields(DataInput in) throws IOException {
        if (first == null) first = new Text();
        if (second == null) second = new Text();
        first.readFields(in);
        second.readFields(in);
    }

    @Override
    public int hashCode() {
        return first.hashCode() * 163 + second.hashCode();
    }
```

```
    @Override
    public boolean equals(Object o) {
        if (o instanceof TextPair) {
            TextPair ip = (TextPair) o;
            return first == ip.first && second == ip.second;
        }
        return false;
    }

    @Override
    public String toString() {
        return first + "\t" + second;
    }

    protected Text getLower() {
        return (compare(first, second) > 0)? second : first;
    }

    protected Text getHigher() {
        return (compare(first, second) > 0)? first : second;
    }

    /**
     * Convenience method for comparing two ints.
     */
    public static int compare(Text a, Text b) {
        return a.compareTo(b);
    }

    public boolean isFirstEmpty() {
        String strFirst = first.toString();
        return strFirst == null || strFirst.isEmpty();
    }

    public boolean isSecondEmpty() {
        String strSecond = second.toString();
        return strSecond == null || strSecond.isEmpty();
    }

    public String getSingleValue() {
        boolean firstEmpty = isFirstEmpty(), secondEmpty =
isSecondEmpty();
        if (firstEmpty && !secondEmpty) return second.toString();
        if (!firstEmpty && secondEmpty) return first.toString();
        return null;
    }

    /* (non-Javadoc)
     * @see java.lang.Comparable#compareTo(java.lang.Object)
     * Compare two TextPair, so that a single pair is compared
regardless of position
     */
```

```
    public int compareTo(TextPair tp) {
        String singleValue = getSingleValue();
        String otherSingleValue = tp.getSingleValue();
        if (singleValue != null && otherSingleValue != null) return
singleValue.compareTo(otherSingleValue);
        String higher = getHigher().toString(), lower =
getLower().toString();
        if (singleValue == null && otherSingleValue != null) {
            if (lower.compareTo(otherSingleValue) < 0) return -1;
            return 1;
        }
        if (singleValue != null && otherSingleValue == null) {
            if (tp.getHigher().toString().compareTo(singleValue) >= 0)
return -1;
            else return 1;
        }
        return compareToBoth(tp);
    }

    public int compareToBoth(TextPair tp) {
        int cmp = compare(first, tp.first);
        if (cmp != 0) {
            return cmp;
        }
        return compare(second, tp.second);
    }

}
```

FUSIONKEYCREATION3.JAVA

```
package fusion.hadoop.fusionkeycreation;

import java.io.IOException;
import java.util.ArrayList;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;

import fusion.hadoop.WordCountFused;
```

```
import fusion.hadoop.fusionexecution.FusionExecution3;

/**
 * Job for creating fusion key pair with list of values associated to
each key
 * this whole list is created to reduce the calls to the actual Mapper
 * and for the fusion key mapping to be available during the fusion
execution map phase
 */
public class FusionKeyCreation3 {
    protected static String JOB_NAME = "FusionKeyCreation3";

    public static class FusionKeyMapper
    extends Mapper<LongWritable, Text, Text, IntWritable> {
        private Text word = new Text();
        private final static IntWritable one = new IntWritable(1);

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {
            ArrayList<String> tokens =
WordCountFused.WordCountMapper.map(value);
            FusionExecution3.FusionExecutionReducer.mapCompute(value,
null);
            for (String token : tokens) {
                word.set(token);
                context.write(word, inputMapper(word));
            }
        }

        public IntWritable inputMapper(Text key)
        {
            return one;
        }
    }

    public static class FusionKeyReducer extends Reducer<Text,
IntWritable, Text, FusionKeysWritable> {
        private Text last = new Text();
        protected IntWritable[] lastValues;
        private boolean lastConsumed = true;
        protected int count = 0;
        protected MultipleOutputs<Text, FusionKeysWritable> outputs;
        protected FusionKeysWritable fusionKeysWritable = new
FusionKeysWritable();
        protected IntWritable[] typeWritableArray = new IntWritable[1];

        @Override
        protected void setup(Context context
                ) throws IOException, InterruptedException {
            outputs = new MultipleOutputs<Text,
FusionKeysWritable>(context);
        }
```

```
        @Override
        public void reduce(Text key, Iterable<IntWritable> values,
                Context context)
                        throws IOException, InterruptedException {
            if (lastConsumed) {
                last.set(key);
                lastValues = createWritableArray(values);
                lastConsumed = false;
            } else {
                lastConsumed = true;
                write(last, lastValues, key,
createWritableArray(values));
            }
        }

        @Override
        protected void cleanup(Context context)
                throws IOException, InterruptedException {
            if (!lastConsumed) {
                write(last, lastValues);
            }
            outputs.close();
        }

        protected void write(Text key, IntWritable[] values) throws
IOException, InterruptedException {
            fusionKeysWritable.Values.set(values);
            fusionKeysWritable.OtherKey = new Text();
            fusionKeysWritable.OtherKey.set("");
            fusionKeysWritable.OtherValues.set(new IntWritable[0]);
            outputs.write(key, fusionKeysWritable, "fusionkeyvalue");
        }

        protected void write(Text last, IntWritable[] lastValues, Text
key, IntWritable[] values) throws IOException, InterruptedException {
            fusionKeysWritable.Values.set(lastValues);
            fusionKeysWritable.OtherKey = key;
            fusionKeysWritable.OtherValues.set(values);
            outputs.write(last, fusionKeysWritable, "fusionkeyvalue");
        }

        protected IntWritable[]
createWritableArray(Iterable<IntWritable> values) {
            ArrayList<IntWritable> listValues = new
ArrayList<IntWritable>();
            for (IntWritable value : values) {
                listValues.add(value);
            }
            IntWritable[] array = new IntWritable[listValues.size()];
            int i=0;
            for (IntWritable value : listValues) {
                array[i++] = value;
```

```
            }
            return array;
        }
    }

    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        main(args[0], args[1]);
    }

    protected static int executeFusionKeyCreationJob(String inputPath,
String outputPath, FileSystem fs) throws IOException,
InterruptedException, ClassNotFoundException {
        System.out.println(JOB_NAME + " job begins");
        Job job = Job.getInstance();
        job.setJarByClass(FusionKeyCreation.class);
        job.setJobName(JOB_NAME);
        FileInputFormat.addInputPath(job, new Path(inputPath));
        FileOutputFormat.setOutputPath(job, new Path(outputPath));
        job.setMapperClass(FusionKeyMapper.class);
        job.setReducerClass(FusionKeyReducer.class);
        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(FusionKeysWritable.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);

        int status = job.waitForCompletion(true) ? 0 : 1;
        System.out.println(JOB_NAME + " job ends with status " +
status);
        /// temporary files are deleted to avoid being read by the next
job
        fs.delete(new Path(outputPath + "/_SUCCESS"), true);
        fs.delete(new Path(outputPath + "/_logs"), true);
        return status;
    }

    public static int main(String inputPath, String outputPath) throws
IOException, InterruptedException, ClassNotFoundException
    {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        fs.delete(new Path(outputPath), true);
        int status = executeFusionKeyCreationJob(inputPath, outputPath,
fs);
        return status;
    }

}
```

FUSIONEXECUTION3.JAVA

```
package fusion.hadoop.fusionexecution;


import java.io.IOException;
import java.net.URISyntaxException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.MultipleOutputs;

import fusion.hadoop.FusionConfiguration;
import fusion.hadoop.TextPair;
import fusion.hadoop.fusionkeycreation.FusionKeyCreation2;
import fusion.hadoop.fusionkeycreation.FusionKeyMap;
import fusion.hadoop.fusionkeycreation.FusionKeyMapParser;
import fusion.hadoop.fusionkeycreation.FusionKeysWritable;


/**
 * Input Reducer from the original word count job by parsing the input
from Job 1
 * Mapper output includes original keys and fused keys
 * which are partitioned into three groups to avoid 2 out of 3 to be
failed
 * in a single process fault
 */
public class FusionExecution3 {

    public static class FusionExecutionMapper
    extends Mapper<Text, FusionKeysWritable, TextPair, IntWritable> {
        protected FusionKeyMapParser kmp;
        protected FusionKeyMap fkm;
        protected TextPair keyPairRaw = new TextPair(), keyPairFused =
new TextPair();
        protected String empty = "";
        protected Text emptyText = new Text();
        @Override
        protected void setup(Context context) throws IOException {
            emptyText.set(empty);
```

```
        }

        @Override
        public void map(Text key, FusionKeysWritable value, Context
context)
                throws IOException, InterruptedException {
            keyPairFused.set(key, value.OtherKey);

            if (value.OtherKey.toString().length() > 0) {
                keyPairRaw.set(emptyText, value.OtherKey);
                Writable[] otherValues = (Writable[])
value.OtherValues.get();
                for (int i=0; i<otherValues.length; ++i) {
                    IntWritable v = (IntWritable) otherValues[i];
                    context.write(keyPairRaw, v);
                    context.write(keyPairFused, v);
                }
            } else {
                keyPairFused.set(key, key);
            }

            keyPairRaw.set(key, emptyText);
            Writable[] values = (Writable[]) value.Values.get();
            for (int i=0; i<values.length; ++i) {
                IntWritable v = (IntWritable) values[i];
                context.write(keyPairRaw, v);
                context.write(keyPairFused, v);
            }
        }
    }


    protected static String jobName = "FusionExecution3";
    public static class FusionExecutionMapper0
        extends Mapper<Text, FusionKeyCreation2.KeyCreationWritable,
TextPair, Writable> {
        protected TextPair keyPairRaw = new TextPair();
        protected Text empty = new Text("");

        @Override
        public void map(Text key,
FusionKeyCreation2.KeyCreationWritable value, Context context)
                throws IOException, InterruptedException {

            keyPairRaw.set(key, empty);
            Writable[] values =((FusionKeyCreation2.ValueArrayWritable)
value.get()).get();
            for (int i=0; i<values.length; ++i) {
                //System.out.println(" mapping: " + key.toString() + "
:: " + values[i]);
                context.write(keyPairRaw, values[i]);
            }
        }
```

```
    }

    public static class FusionExecutionMapper1
    extends Mapper<Text, FusionKeyCreation2.KeyCreationWritable,
TextPair, Writable> {
        protected TextPair keyPairRaw = new TextPair();
        protected Text empty = new Text("");

        @Override
        public void map(Text key,
FusionKeyCreation2.KeyCreationWritable value, Context context)
                throws IOException, InterruptedException {

            keyPairRaw.set(empty, key);
            Writable[] values =((FusionKeyCreation2.ValueArrayWritable)
value.get()).get();
            for (int i=0; i<values.length; ++i) {
                //System.out.println(" mapping: " + key.toString() + "
:: " + values[i]);
                context.write(keyPairRaw, values[i]);
            }
        }
    }


    public static class FusionExecutionReducer
    extends Reducer<TextPair, IntWritable, Text, IntWritable> {

        private Text fusedKey = new Text();
        private MultipleOutputs<Text, IntWritable> multipleOutputs;

        @Override
        protected void setup(Context context)
                throws IOException, InterruptedException {
            multipleOutputs = new MultipleOutputs<Text,
IntWritable>(context);
        }

        @Override
        public void reduce(TextPair key, Iterable<IntWritable> values,
                Context context)
                        throws IOException, InterruptedException {
            if (key.getSecond().toString().length() == 0) {
                /// write to raw key result
                IntWritable value = inputReduce(key.getFirst(),
values);
                if (value != null)
multipleOutputs.write(key.getFirst(), value, "result");
            } else if (key.getFirst().toString().length() == 0) {
                IntWritable value = inputReduce(key.getSecond(),
values);
                if (value != null)
multipleOutputs.write(key.getSecond(), value, "result");
```

```
            } else {
                fusedKey.set(key.toString());
                IntWritable value = inputReduce(fusedKey, values);
                if (value != null) {
                    if
(key.getFirst().toString().compareTo(key.getSecond().toString()) != 0)
{
                        multipleOutputs.write(fusedKey, value,
"fused_result");
                    } else {
                        /// single key with no other key
                        multipleOutputs.write(key.getSecond(), value,
"fused_result");
                    }
                }
            }
        }

        protected IntWritable inputReduce(Text key,
Iterable<IntWritable> values) {
            try {
                int sum = 0;
                /// simulation for application fault
                /// if (key.toString().compareTo("a") == 0) throw new
Exception("Fail to reduce.");
                for (IntWritable value : values) {
                    sum += value.get();
                }
                reduceCompute(key, values);
                return new IntWritable(sum);
            } catch (Exception ex) {
                System.err.println("Error when reducing key: " +
key.toString() + "\n\t" + ex.getMessage());
            }
            return null;
        }


        protected static int reduceComplexity = 0;
        public static void reduceCompute(Text key,
Iterable<IntWritable> values) {
            ///simulate long running process
            for (int k=0; k<reduceComplexity; ++k) {
                for (int i=2001; i<2999; ++i) {
                    boolean isPrime = true;
                    for (int j=2; j<i; ++j) {
                        if ((i % j) == 0) {
                            isPrime = false;
                            break;
                        }
                    }
                    if (isPrime) { }
                }
```

```
            }
        }

        protected static int mapComplexity = 0;
        public static void mapCompute(Text key, Iterable<IntWritable>
values) {
            ///simulate long running process
            for (int k=0; k<mapComplexity; ++k) {
                for (int i=2001; i<2999; ++i) {
                    boolean isPrime = true;
                    for (int j=2; j<i; ++j) {
                        if ((i % j) == 0) {
                            isPrime = false;
                            break;
                        }
                    }
                    if (isPrime) { }
                }
            }
        }

        @Override
        protected void cleanup(Context context)
                throws IOException, InterruptedException {
            multipleOutputs.close();
        }
    }

    public static class Partitioner extends FusionPartitioner {
        public Partitioner() {
            super(FusionConfiguration.NUM_OF_REDUCERS);
        }
    }

    protected static int executeFusionExecutionJob(String inputPath,
String outputPath, FileSystem fs) throws IOException,
InterruptedException, ClassNotFoundException, URISyntaxException {
        System.out.println(jobName + " job begins");
        Configuration conf = new Configuration();
        conf.setInt("mapreduce.reduce.maxattempts", 1);
        conf.setInt("mapred.reduce.max.attempts", 1);
        conf.setBoolean("mapred.reduce.tasks.speculative.execution",
false);
        conf.setInt("mapred.max.reduce.failures.percent", 49);
        conf.setInt("mapreduce.job.reduces",
FusionConfiguration.NUM_OF_REDUCERS);
        Job job = Job.getInstance(conf);
        job.setJarByClass(FusionExecution.class);
        job.setJobName("FusionExecution");

        FileInputFormat.addInputPath(job, new Path(inputPath));
        FileOutputFormat.setOutputPath(job, new Path(outputPath));
        job.setInputFormatClass(SequenceFileInputFormat.class);
```

```
        job.setMapperClass(FusionExecutionMapper.class);
        job.setReducerClass(FusionExecutionReducer.class);
        job.setNumReduceTasks(FusionConfiguration.NUM_OF_REDUCERS);
        job.setPartitionerClass(Partitioner.class);

        job.setMapOutputKeyClass(TextPair.class);
        job.setMapOutputValueClass(IntWritable.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(IntWritable.class);

        int status = job.waitForCompletion(true) ? 0 : 1;
        System.out.println(jobName +  " job ends with status " +
status);
        return status;
    }

    public static int main(String inputPath, String outputPath) throws
IOException, InterruptedException, ClassNotFoundException,
URISyntaxException
    {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        fs.delete(new Path(outputPath), true);
        int status = executeFusionExecutionJob(inputPath +
"/fusionkeyvalue*", outputPath, fs);
        return status;
    }

    public static void main( String[] args ) throws IOException,
InterruptedException, ClassNotFoundException
    {
        System.out.println("\n*** compute start...");
        long msStart = System.currentTimeMillis();

        FusionExecutionReducer.reduceCompute(null, null);

        long msEnd = System.currentTimeMillis();
        System.out.println("\n*** Total elapsed: " + (msEnd - msStart)
+ "ms");
    }

}
```

FUSIONPARTITIONER.JAVA

```
package fusion.hadoop.fusionexecution;

import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.mapreduce.Partitioner;

import fusion.hadoop.TextPair;
```

```
/**
 * Partition keys into three buckets, 1 for fused key, and two for its
parent
 * Number of partition is assumed to be multiples of 3
 */
public class FusionPartitioner extends Partitioner<TextPair,
IntWritable> {

    protected static int BUCKET_COUNT = 3;
    protected int[] bucketSize;
    protected int[] bucketOffst;

    public FusionPartitioner(int numPartitions) {
        bucketSize = createBucketSize(numPartitions);
        bucketOffst = createBucketOffset(bucketSize);
    }

    @Override
    public int getPartition(TextPair key, IntWritable value, int
numPartitions) {
        int bucketIdx = getBucketIdx(key);
        return getAssignedPartition(key, bucketIdx);
    }

    protected int getAssignedPartition(TextPair key, int bucketIdx) {
        int size = bucketSize[bucketIdx];
        return bucketOffst[bucketIdx] + ((key.getFirst().hashCode() %
size) + (key.getSecond().hashCode() % size)) % size;
    }

    protected int getBucketIdx(TextPair key) {
        String first = key.getFirst().toString(), second =
key.getSecond().toString();
        if (first.length() > 0 && second.length() > 0) return 0;
        return (first.length() > 0)? 1 : 2;
    }

    protected int[] createBucketOffset(int[] bucketSize) {
        int[] bucketOffset = new int[BUCKET_COUNT];
        bucketOffset[0] = 0;
        for (int i=0; i<BUCKET_COUNT-1; ++i) {
            bucketOffset[i+1] = bucketSize[i] + bucketOffset[i];
        }
        return bucketOffset;
    }

    protected int[] createBucketSize(int numPartitions) {
        int[] bucketSizes = new int[BUCKET_COUNT];
        int averageSize = numPartitions / BUCKET_COUNT;
        bucketSizes[0] = averageSize;
        bucketSizes[1] = averageSize;
        bucketSizes[2] = averageSize;
        int remainder = numPartitions % BUCKET_COUNT;
```

```
        for (int i=1, j=0; i<=remainder; ++i, j=(j+1)%BUCKET_COUNT) {
            ++bucketSizes[j];
        }
        return bucketSizes;
    }
}
```

MISSEDKEYSEARCH3.JAVA

```
package fusion.hadoop.missedKeySearch;

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.lib.output.SequenceFileOutputFormat;

import fusion.hadoop.fusionkeycreation.FusionKeyCreation;
import fusion.hadoop.fusionkeycreation.FusionKeysWritable;

/**
 * Search for keys missing in result by utilizing fusion keys created
in Job 1
 */
public class MissedKeySearch3 {

    public static class EmptyTextMapper
    extends Mapper<LongWritable, Text, Text, Text> {
        private Text word = new Text();
        private final static Text emptyText = new Text("");

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {

            String[] keys = value.toString().split("\t");
            if (keys.length > 0) {
                word.set(keys[0]);
                context.write(word, emptyText);
            }
        }
    }
```

```
    public static class FusionKeyValueMapper
    extends Mapper<Text, FusionKeysWritable, Text, Text> {

        @Override
        public void map(Text key, FusionKeysWritable value, Context
context)
                throws IOException, InterruptedException {
            if (value.OtherKey.toString().length() > 0) {
                context.write(key, value.OtherKey);
                context.write(value.OtherKey, key);
            } else {
                context.write(key, key);
            }
        }
    }


    public static class keyPairMapper
    extends Mapper<LongWritable, Text, Text, Text> {

        private Text key1 = new Text(), key2 = new Text(), value1 = new
Text(), value2 = new Text();

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {

            String[] keys = value.toString().split("\t");
            if (keys.length > 1) {
                key1.set(keys[0]);
                value1.set(keys[1]);
                key2.set(keys[1]);
                value2.set(keys[0]);
                context.write(key1, value1);
                context.write(key2, value2);
            } else {
                key1.set(keys[0]);
                value1.set(keys[0]);
                context.write(key1, value1);
            }
        }
    }

    public static class MissedKeySearchReducer
    extends Reducer<Text, Text, Text, Text> {

        private Text recoveryKeyText = new Text();
        private int missingKeyCount = 0;
        @Override
        public void reduce(Text key, Iterable<Text> values,
                Context context)
                        throws IOException, InterruptedException {
```

```
            boolean emptyExists = false;
            String recoveryKey = null;
            for (Text value : values) {
                String keyString = value.toString();
                if (keyString.length() == 0) emptyExists = true;
                else recoveryKey = keyString;
            }

            if (!emptyExists && recoveryKey != null) {
                System.out.println("\trecovery keys: " + key + "\t" +
key.toString().length());
                ++missingKeyCount;
                recoveryKeyText.set(recoveryKey);
                context.write(recoveryKeyText, key);
            }
        }
    }

    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException {
        main(args[0], args[1], args[2]);
    }


    protected static int executeMissedKeySearchJob(String resultPath,
String fusedKeyPath, String outputPath, FileSystem fs) throws
IOException, InterruptedException, ClassNotFoundException {

        System.out.println("MissedKeySearch3 job begins");
        Job job = Job.getInstance();
        job.setJarByClass(FusionKeyCreation.class);
        job.setJobName("MissedKeySearch");

        FileOutputFormat.setOutputPath(job, new Path(outputPath));

        MultipleInputs.addInputPath(job, new Path(resultPath),
TextInputFormat.class, EmptyTextMapper.class);
        MultipleInputs.addInputPath(job, new Path(fusedKeyPath),
SequenceFileInputFormat.class, FusionKeyValueMapper.class);
        job.setReducerClass(MissedKeySearchReducer.class);
        job.setOutputFormatClass(SequenceFileOutputFormat.class);

        job.setMapOutputKeyClass(Text.class);
        job.setMapOutputValueClass(Text.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(Text.class);

        int status = job.waitForCompletion(true) ? 0 : 1;
        System.out.println("MissedKeySearch3 job ends with status " +
status);
        return status;
    }
```

```
    public static int main(String resultPath, String fusedKeyPath,
String outputPath) throws IOException, InterruptedException,
ClassNotFoundException
    {
        Configuration conf = new Configuration();
        FileSystem fs = FileSystem.get(conf);
        fs.delete(new Path(outputPath), true);

        int status = executeMissedKeySearchJob(resultPath,
fusedKeyPath, outputPath, fs);

        return status;
    }
}
```

DEFUSEMISSEDKEYSWITHOUTMAPFILE.JAVA

```
package fusion.hadoop.defuseMissedKeys;

import java.io.IOException;
import java.net.URISyntaxException;

import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.fs.FileStatus;
import org.apache.hadoop.fs.FileSystem;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.ArrayWritable;
import org.apache.hadoop.io.GenericWritable;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.mapreduce.Job;
import org.apache.hadoop.mapreduce.Mapper;
import org.apache.hadoop.mapreduce.Reducer;
import org.apache.hadoop.mapreduce.lib.input.MultipleInputs;
import org.apache.hadoop.mapreduce.lib.input.SequenceFileInputFormat;
import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;

import fusion.hadoop.TextPair;

/**
 * Recover missed key from fusion result, original result, and missed
key result
 */
public class DefuseMissedKeysWithoutMapFile {

    public static class DefuseMissedKeyWritable extends GenericWritable
{

        @SuppressWarnings("rawtypes")
        private static Class[] CLASSES = {
```

```
            DefuseArrayWritable.class,
            Text.class
        };

        @SuppressWarnings({ "unchecked", "rawtypes" })
        protected Class[] getTypes() {
            return CLASSES;
        }

    }

    public static class MissingKeyMapper extends Mapper<Text, Text,
Text,DefuseMissedKeyWritable> {
        protected DefuseMissedKeyWritable defuseMissedKeyWritable= new
DefuseMissedKeyWritable();

        @Override
        public void map(Text key, Text value, Context context)
                throws IOException, InterruptedException {
            defuseMissedKeyWritable.set(value);
            context.write(key, defuseMissedKeyWritable);
        }
    }

    private static Class<? extends Writable> VALUE_CLASS =
IntWritable.class;
    public static class DefuseMapper2_1 extends DefuseMapper {
        public DefuseMapper2_1() {
            super(2, 1);
        }
    }

    public static class DefuseMapper
    extends Mapper<LongWritable, Text, Text, DefuseMissedKeyWritable> {
        private Text word = new Text();
        protected int sourceKeyCount = 1;
        protected int targetIndex = 0;

        protected TextPair keyPairRaw = new TextPair(), keyPairFused =
new TextPair();
        protected DefuseArrayWritable values = new
DefuseArrayWritable();
        protected DefuseMissedKeyWritable missedKeyValue = new
DefuseMissedKeyWritable();

        public DefuseMapper(int sourceKeyCount, int targetIndex) {
            super();
            this.targetIndex = targetIndex;
            this.sourceKeyCount = sourceKeyCount;
        }

        public DefuseMapper() {
            super();
```

```
        }

        @Override
        public void map(LongWritable key, Text value, Context context)
                throws IOException, InterruptedException {

            String[] keys = value.toString().split("\t");
            IntWritable[] valueArray = new IntWritable[targetIndex +
1];
            int i=0;
            while (i<targetIndex) {
                valueArray[i] = new IntWritable();
                ++i;
            }
            if (keys.length > 1 && keys.length > sourceKeyCount) {
                String fusedResult = keys[sourceKeyCount];
                for (i=0; i<sourceKeyCount; ++i) {
                    String keyString = keys[i];
                    word.set(keyString);
                    valueArray[targetIndex] = new
IntWritable(Integer.parseInt(fusedResult));
                    values.set(valueArray);
                    missedKeyValue.set(values);
                    context.write(word, missedKeyValue);
                }
            }
        }
    }

    public static class DefuseReducer
    extends Reducer<Text, DefuseMissedKeyWritable, Text, IntWritable> {

        @Override
        public void reduce(Text key, Iterable<DefuseMissedKeyWritable>
values,
                Context context)
                        throws IOException, InterruptedException {
            IntWritable fusedResult = null;
            IntWritable recoveryResult = null;
            Text missingKey = null;
            for (DefuseMissedKeyWritable defuseMissingKeyWritable :
values) {
                Writable writable = defuseMissingKeyWritable.get();
                if (writable instanceof Text) missingKey = (Text)
writable;
                else if (writable instanceof DefuseArrayWritable){
                    DefuseArrayWritable value =  (DefuseArrayWritable)
writable;
                    if (value.get().length > 1) fusedResult =
(IntWritable) value.get()[1];
                    else recoveryResult = (IntWritable) value.get()[0];
                }
            }
```

```
            if (fusedResult != null && recoveryResult != null &&
missingKey != null) {
                /// keys missing in result
                context.write(missingKey, Defuse(fusedResult,
recoveryResult));
            }
        }
    }

    public static class DefuseArrayWritable extends ArrayWritable {
        public DefuseArrayWritable() {
            super(VALUE_CLASS);
        }
    }

    public static IntWritable Defuse(IntWritable fusedResult,
IntWritable recoveryResult) {
        IntWritable defuseResult = new IntWritable();
        defuseResult.set(fusedResult.get() - recoveryResult.get());
        return defuseResult;
    }

    public static void main(String[] args) throws IOException,
InterruptedException, ClassNotFoundException, URISyntaxException {
        main(args[0], args[1], args[2], args[3]);
    }

    protected static int addMissedKeyCacheFiles(Configuration job,
FileSystem fs, String missingKeyPath) throws IOException,
URISyntaxException {
        String pattern = missingKeyPath + "/part-r-*";
        int fileCount = 0;

        FileStatus[] fss = fs.globStatus(new Path(pattern));
        for (FileStatus fst : fss) {
            if (fst.getLen() > 0) {
                //DistributedCache.addCacheFile(fst.getPath().toUri(),
job);
                System.out.println("\tadding cache path: " +
fst.getPath().toString());
                ++fileCount;
            }
        }
        return fileCount;
    }

    protected static int executeDefuseMissedKeysJob(String resultPath,
String fusedResultPath, String missingKeyPath, String
missingKeyResultPath, FileSystem fs) throws IOException,
InterruptedException, ClassNotFoundException, URISyntaxException {
        int status = 0;
        System.out.println("DefuseMissedKey job begins");
```

```
        Configuration conf = new Configuration();
        if (addMissedKeyCacheFiles(conf, fs, missingKeyPath) > 0) {
            Job job = Job.getInstance(conf);
            job.setJarByClass(DefuseMissedKeys.class);
            job.setJobName("DefuseMissedKey");
            FileOutputFormat.setOutputPath(job, new
Path(missingKeyResultPath));

            MultipleInputs.addInputPath(job, new Path(missingKeyPath),
SequenceFileInputFormat.class, MissingKeyMapper.class);
            MultipleInputs.addInputPath(job, new Path(resultPath),
TextInputFormat.class, DefuseMapper.class);
            MultipleInputs.addInputPath(job, new Path(fusedResultPath),
TextInputFormat.class, DefuseMapper2_1.class);

            job.setReducerClass(DefuseReducer.class);

            job.setMapOutputKeyClass(Text.class);
            job.setMapOutputValueClass(DefuseMissedKeyWritable.class);
            job.setOutputKeyClass(Text.class);
            job.setOutputValueClass(IntWritable.class);

            status = job.waitForCompletion(true) ? 0 : 1;
            System.out.println("DefuseMissedKey job ends with status "
+ status);
        } else {
            System.out.println("DefuseMissedKey job skipped due to
empty missed key files.");
        }
        return status;
    }

    public static int main(String resultPath, String fusedResultPath,
String missingKeyPath, String missingKeyResultPath) throws IOException,
InterruptedException, ClassNotFoundException, URISyntaxException
    {
        Configuration conf = new Configuration();
        // configuration should contain reference to your namenode
        FileSystem fs = FileSystem.get(conf);
        // true stands for recursively deleting the folder you gave
        fs.delete(new Path(missingKeyResultPath), true);

        int status = executeDefuseMissedKeysJob(resultPath,
fusedResultPath, missingKeyPath, missingKeyResultPath, fs);

        return status;
    }
}
```

References

[1] B. Balasubramanian and V. K. Garg. Fusion-based DFSMs for Fault Tolerance in Distributed Systems, Parallel and Distributed Systems Laboratory, ECE Dept. University of Texas at Austin, 2011.

[2] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. Google, Inc. 2004.

[3] Ghemawat, Sanjay; Gobioff, Howard; Leung, Shun-Tak, The Google File System, 19th Symposium on Operating Systems Principles (conference), Lake George, NY: The Association for Computing Machinery, CiteSeerX: 10.1.1.125.789, 2012

[4] The Apache Software Foundation. Welcome to Apache™ Hadoop®! What Is Apache Hadoop? http://hadoop.apache.org/?page=2#What+Is+Apache+Hadoop%3F 2012

[5] Yahoo! Inc. Apache Hadoop Module 4: MapReduce 6.3 Fault Tolerance http://developer.yahoo.com/hadoop/tutorial/module4.html#tolerence

[6] Tom White. Hadoop: The Definitive Guide. 3rd Edition, pages 202-203, 2012

[7] Bharath Balasubramanian, Vijay K. Garg: Fault Tolerance in Distributed Systems Using Fused Data Structures. IEEE Trans. Parallel Distrib. Syst. 24 (4): 701-715 (2013)

[8] Bharath Balasubramanian, Vijay K. Garg: Fused Data Structures for Handling Multiple Faults in Distributed Systems. ICDCS 2011 : 677-688